Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

An LLM-Based Description Generation System for Enterprise GitHub Repositories

Author: Chenqi Xie (2757089)

1st supervisor:	Adam Belloum	
daily supervisor:	Alwin den Herder	(Arcadis)
2nd reader:	Natallia Kokash	

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

June 29, 2025

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Abstract

Context. Many GitHub repositories, especially in large organizations, lack clear or complete descriptions. This reduces the discoverability and usability of valuable internal code resources, as developers must manually inspect repository contents to understand their purpose.

Goal. This study aims to develop a system that automatically generates concise and informative descriptions for GitHub repositories by leveraging large language models (LLMs). The system is designed to support internal documentation and repository indexing at scale.

Method. A three-stage pipeline is proposed: metadata extraction, data preprocessing, and prompt-based description generation using GPT-4o. Prompts are designed based on the LSP (Language, Software, Purpose) framework, extended with a user dimension. The system is evaluated on 330 repositories using BERTScore to assess semantic alignment.

Results. The LLM-generated descriptions outperform both rule-based and random baselines in BERTScore F1. In addition, the generated outputs show strong stylistic and semantic similarity to human-written descriptions, suggesting that the model captures key repository information effectively.

Conclusions. The findings demonstrate that LLMs can be used to generate developer-friendly summaries from repository metadata. This approach has practical value for organizations seeking to improve documentation coverage and support solution recommendation or onboarding workflows.

Contents

CONTENTS

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

1

Introduction

GitHub is one of the most widely used platforms in software development and project management area. Currently, there are over 420 million repositories stored on GitHub(1). These repositories contain source code, resources, and other related informations. When users browse a repository, they can see the repository name, owner, a short description, main programming language, and contributors. The description field is the only part that users fill out when creating a repository and provides the first impression of the repository's purpose. Since the description is optional (see Figure 1.1), many repositories have descriptions that are too brief or unclear, or even left empty. This forces developers to spend time exploring the repository's content or reading the code to understand what the repository does. It becomes difficult for developers to determine if a repository fits their needs when search a solution on Github.

With the volume of Github repositories being so large, manually updating and maintaining descriptions for every repository is not feasible. Thus, developing a pipeline that can automatically generate descriptions is very helpful and essential. To address this problem, Large Language Models (LLMs) shows great potential. LLMs, which are pre-trained models designed to understand and generate human-like text. In various natural language processing (NLP) tasks, such as text generation, translation and summarization, LLMs have shown remarkable performance(2). LLMs are particularly useful when generating text that requires both understanding of the context and coherence in output, making them ideal for tasks like summary generation. Traditional methods like rule-based machine learning systems or manual writing are time-consuming and often inconsistent, especially when scaling to large datasets.

When applied to GitHub repository analysis, LLMs can extract key information based on repository's metadata, code files, documentation, and other resources, and automatically

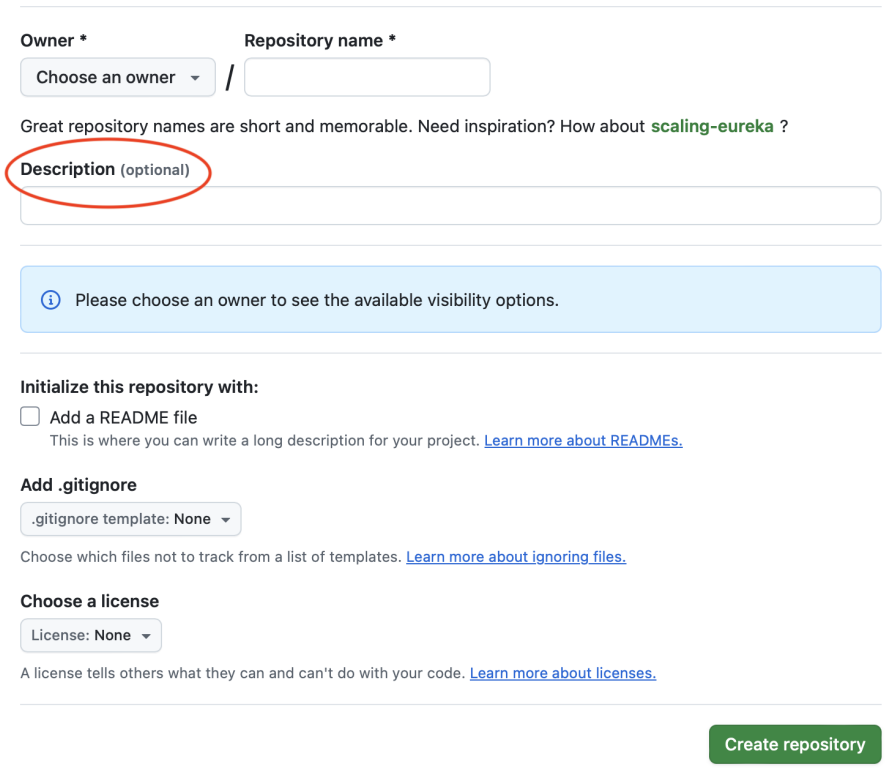

Figure 1.1: Example of GitHub repository creation page

generate accurate descriptions. This automated description generation not only improves efficiency but also produces more precise and meaningful descriptions for each repository. This improves developers' efficiency and helps maximize the use of resources on GitHub.

In previous research on GitHub repositories, much of the focus has been on repository classification and recommendation(3)(4)(5)(6), or on the analysis of specific projects within certain domains of computer science, such as Deep Learning(7)(8) or Data Science(9)(10). However, there has been limited research specifically addressing repository descriptions. For example, Hellman et al. (2021) primarily focus on comparing manual and automated approaches to generating repository descriptions (11). With the advancement of large language models (LLMs), there is now a growing opportunity to leverage these models for automating the generation of more precise and useful descriptions for a wide range of repositories. This research aims to bridge this gap by exploring a method that not only automates the generation of repository descriptions using LLMs, but also addresses the challenges in retrieving and structuring relevant data from repositories to enhance description quality.

Towards this direction, we propose a system that extracts structured metadata from repositories and uses an LLM to generate descriptive summaries.

This study is guided by the following research questions:

- **RQ1.** How to extract and refine metadata fields to effectively capture the essential characteristics of a repository?
- **RQ2.** How to automatically generate precise and useful repository descriptions by LLM?

To answer these questions, we designed and implemented a comprehensive pipeline that enables the automatic retrieval of all repositories within a specific organization. It extracts relevant data from these repositories, which is then preprocessed to ensure quality and structure. We also developed an efficient prompt for generating repository descriptions that meet the required standards. Using Azure GPT-4o, we generated descriptions for repositories based on this prompt. In our experiments, we applied this approach to a randomly sampled set of 330 repositories that already had descriptions. We quantitatively evaluated the generated outputs using BERTScore and compared them against baselines. The results help assess both the feasibility and the quality of LLM-based repository summarization.

This research makes the following contributions:

1. A scalable metadata extraction pipeline for GitHub repositories, capable of retrieving structural, textual, and historical information in a token-aware and batch-friendly format suitable for LLM consumption.

2. A prompt-based description generation method using GPT-4o, designed to produce concise and semantically informative repository summaries based on structured input metadata.

3. An evaluation framework combining LLM-generated outputs, repository metadata, and human-written descriptions, assessed through BERTScore.

The remainder of this paper is organized as follows. Section 2 provides the Background, including an overview of GitHub, large language models, and prompt engineering techniques. Section 3 reviews the Related Work in the field, highlighting previous research on GitHub repositories and description generation methods. In Section 4, the Methodology is described in detail, covering the system architecture, data extraction process, data preprocessing techniques, and the description generation process using large language models.

Section 5 outlines the Evaluation, including the metrics used for assessment and the experimental setup. The Results of our experiments are presented in Section 6, followed by a Discussion in Section 7. Finally, Section 8 concludes the paper and outlines potential future research directions.

2

Background

2.1 Github

GitHub as a platform Version control plays an important role in software engineering, helping software engineers track and maintain changes to source code, configuration files, and other files, etc. With the development of version control systems (VCS)(12), there are many VCS programs exist. Introduced in 2005, Git is currently the preferred Distributed Version Control System (DVCS) among developers because of its flexibility and free for use(13). By using Git, each developer has a full copy of the repository on their local machine, so they can commit changes independently. GitHub is a cloud-based hosting and collaboration platform for developers to store and manage Git repositories online, Figure 2.1 shows how it works. It allows developers to easily access, update, and manage project files anytime, anywhere. It providing collaboration tools such as pull requests for suggesting changes, issue tracking for managing bugs and feature requests, and project management tools for organizing tasks and tracking progress. Github also integrates continuous integration and deployment (CI/CD) tools that can automatically test, build, and deploy. This can help teams streamline the development process and ensure that code changes do not conflict with existing environment. Additionally, GitHub is the birthplace of numerous open source projects. There is a strong community where developers can contribute and collaborate to public repositories, ultimately enhancing the entire software development ecosystem.

GitHub Repository A repository (or repo) is a storage location where files and their revision history are kept. In software development, repositories typically contain a project's source code as well as other necessary files such as configuration files, documentation,

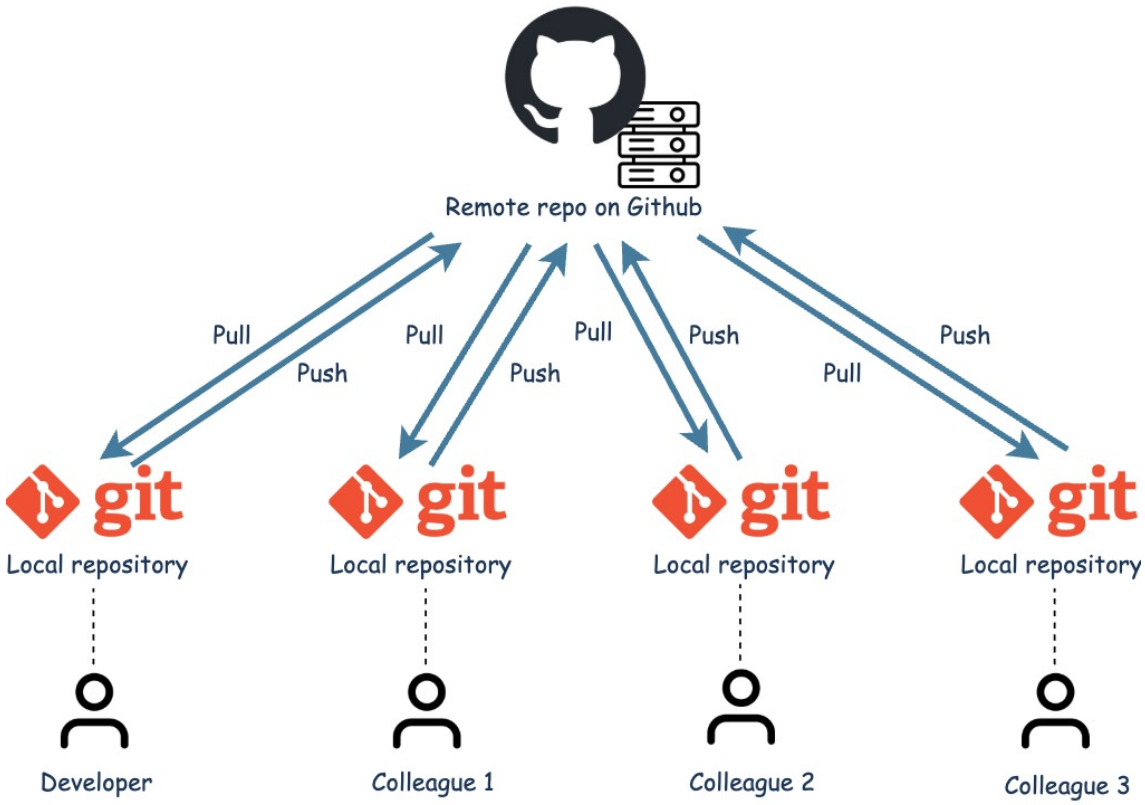

Figure 2.1: Git and Github

and assets such as images or datasets. GitHub repositories are repositories hosted on platforms such as GitHub. Unlike local repositories, which exist only on a developer's personal computer, remote repositories on GitHub allow team GitHub repositories are both a storage system and a collaboration tool to help manage a project's code base over time.

GitHub repositories contain a variety of metadata that can be accessed through the GitHub API. These metadata fields provide basic information about this repository, its contents, and activities. Key fields include:

- **Basic Information:** Repository ID, name, description, and URL.
- **Owner Information:** GitHub username, profile URL. This field can help contextualize the repository within the organization or the individual developer's portfolio.
- **Repository Activity:** Fork status, size, stars, and open issues count.
- **Collaboration:** Number of forks, contributors, and associated topics.

These fields provide essential insights into a repository's content, activity, and collaboration. To better illustrate how this data is structured, the following screenshot shows a sample of the fields that can be retrieved from the GitHub API.

```
[
  {
    "id": 1296269,
    "node_id": "MDEwOlJlcG9zaXRvcnkxMjk2MjY5",
    "name": "Hello-World",
    "full_name": "octocat/Hello-World",
    "owner": {
      "login": "octocat",
      "id": 1,
      "node_id": "MDQ6VXNlcjE=",
      "avatar_url": "https://github.com/images/error/octocat_happy.gif",
      "gravatar_id": "",
      "url": "https://api.github.com/users/octocat",
      "html_url": "https://github.com/octocat",
      "followers_url": "https://api.github.com/users/octocat/followers",
      "following_url": "https://api.github.com/users/octocat/following{/other_user}",
      "gists_url": "https://api.github.com/users/octocat/gists{/gist_id}",
      "starred_url": "https://api.github.com/users/octocat/starred{/owner}{/repo}",
      "subscriptions_url": "https://api.github.com/users/octocat/subscriptions",
      "organizations_url": "https://api.github.com/users/octocat/orgs",
      "repos_url": "https://api.github.com/users/octocat/repos",
      "events_url": "https://api.github.com/users/octocat/events{/privacy}",
      "received_events_url": "https://api.github.com/users/octocat/received_events",
      "type": "User",
      "site_admin": false
    },
    "private": false,
    "html_url": "https://github.com/octocat/Hello-World",
    "description": "This your first repo!",
    "fork": false,
    "url": "https://api.github.com/repos/octocat/Hello-World",
    "archive_url": "https://api.github.com/repos/octocat/Hello-World/{archive_format}{/ref}",
    "assignees_url": "https://api.github.com/repos/octocat/Hello-World/assignees{/user}",
    "blobs_url": "https://api.github.com/repos/octocat/Hello-World/git/blobs{/sha}",
    "branches_url": "https://api.github.com/repos/octocat/Hello-World/branches{/branch}",
    "collaborators_url": "https://api.github.com/repos/octocat/Hello-World/collaborators{/coll
    "comments_url": "https://api.github.com/repos/octocat/Hello-World/comments{/number}",
    "commits_url": "https://api.github.com/repos/octocat/Hello-World/commits{/sha}",
    "compare_url": "https://api.github.com/repos/octocat/Hello-World/compare/{base}...{head}",
    "contents_url": "https://api.github.com/repos/octocat/Hello-World/contents/{+path}",
    "contributors_url": "https://api.github.com/repos/octocat/Hello-World/contributors",
    "deployments_url": "https://api.github.com/repos/octocat/Hello-World/deployments",
    "downloads_url": "https://api.github.com/repos/octocat/Hello-World/downloads",
    "events_url": "https://api.github.com/repos/octocat/Hello-World/events",
    "forks_url": "https://api.github.com/repos/octocat/Hello-World/forks",
    "git_commits_url": "https://api.github.com/repos/octocat/Hello-World/git/commits{/sha}",
    "git_refs_url": "https://api.github.com/repos/octocat/Hello-World/git/refs{/sha}",
    "git_tags_url": "https://api.github.com/repos/octocat/Hello-World/git/tags{/sha}",
    "git_url": "git:github.com/octocat/Hello-World.git",
    "issue_comment_url": "https://api.github.com/repos/octocat/Hello-World/issues/comments{/nu
    "issue_events_url": "https://api.github.com/repos/octocat/Hello-World/issues/events{/numbe
    "issues_url": "https://api.github.com/repos/octocat/Hello-World/issues{/number}",
    "keys_url": "https://api.github.com/repos/octocat/Hello-World/keys{/key_id}",
    "labels_url": "https://api.github.com/repos/octocat/Hello-World/labels{/name}",
    "languages_url": "https://api.github.com/repos/octocat/Hello-World/languages",
    "merges_url": "https://api.github.com/repos/octocat/Hello-World/merges",
    "milestones_url": "https://api.github.com/repos/octocat/Hello-World/milestones{/number}",
    "notifications_url": "https://api.github.com/repos/octocat/Hello-World/notifications{?sinc
    "pulls_url": "https://api.github.com/repos/octocat/Hello-World/pulls{/number}",
    "releases_url": "https://api.github.com/repos/octocat/Hello-World/releases{/id}",
    "ssh_url": "git@github.com:octocat/Hello-World.git",
    "stargazers_url": "https://api.github.com/repos/octocat/Hello-World/stargazers",
    "statuses_url": "https://api.github.com/repos/octocat/Hello-World/statuses/{sha}",
    "subscribers_url": "https://api.github.com/repos/octocat/Hello-World/subscribers",
    "subscription_url": "https://api.github.com/repos/octocat/Hello-World/subscription",
    "tags_url": "https://api.github.com/repos/octocat/Hello-World/tags",
    "teams_url": "https://api.github.com/repos/octocat/Hello-World/teams",
    "trees_url": "https://api.github.com/repos/octocat/Hello-World/git/trees{/sha}",
    "clone_url": "https://github.com/octocat/Hello-World.git",
    "mirror_url": "git:git.example.com/octocat/Hello-World",
    "hooks_url": "https://api.github.com/repos/octocat/Hello-World/hooks",
    "svn_url": "https://svn.github.com/octocat/Hello-World",
    "homepage": "https://github.com",
    "language": null,
    "forks_count": 9,
    "stargazers_count": 80,
    "watchers_count": 80,
    "size": 108,
    "default_branch": "master",
    "open_issues_count": 0,
    "is_template": false,
    "topics": [
      "octocat",
      "atom",
      "electron",
      "api"
    ],
    "has_issues": true,
    "has_projects": true,
    "has_wiki": true,
    "has_pages": false,
    "has_downloads": true,
    "has_discussions": false,
    "archived": false,
    "disabled": false,
    "visibility": "public",
    "pushed_at": "2011-01-26T19:06:43Z",
    "created_at": "2011-01-26T19:01:12Z",
    "updated_at": "2011-01-26T19:14:43Z",
    "permissions": {
      "admin": false,
      "push": false,
      "pull": true
    },
    "security_and_analysis": {
      "advanced_security": {
        "status": "enabled"
      },
      "secret_scanning": {
        "status": "enabled"
      },
      "secret_scanning_push_protection": {
        "status": "disabled"
      },
      "secret_scanning_non_provider_patterns": {
        "status": "disabled"
      }
    }
  }
]
```


Figure 2.2: Sample of the fields retrieved from the GitHub API

2.2 Large Language Models

The concept of language models exists since the 1990s. These models are designed to predict or generate linguistic units, such as words, phrases, or sentences.(14). With the evolution of language models, there are four types have introduced, including Statistical Language Models, Neural Language Models, Pre-trained Language Models, and Large Language Models. Figure 2.3 illustrates the stages of language model development.

The development of language models reached its fourth stage with the birth of large-scale

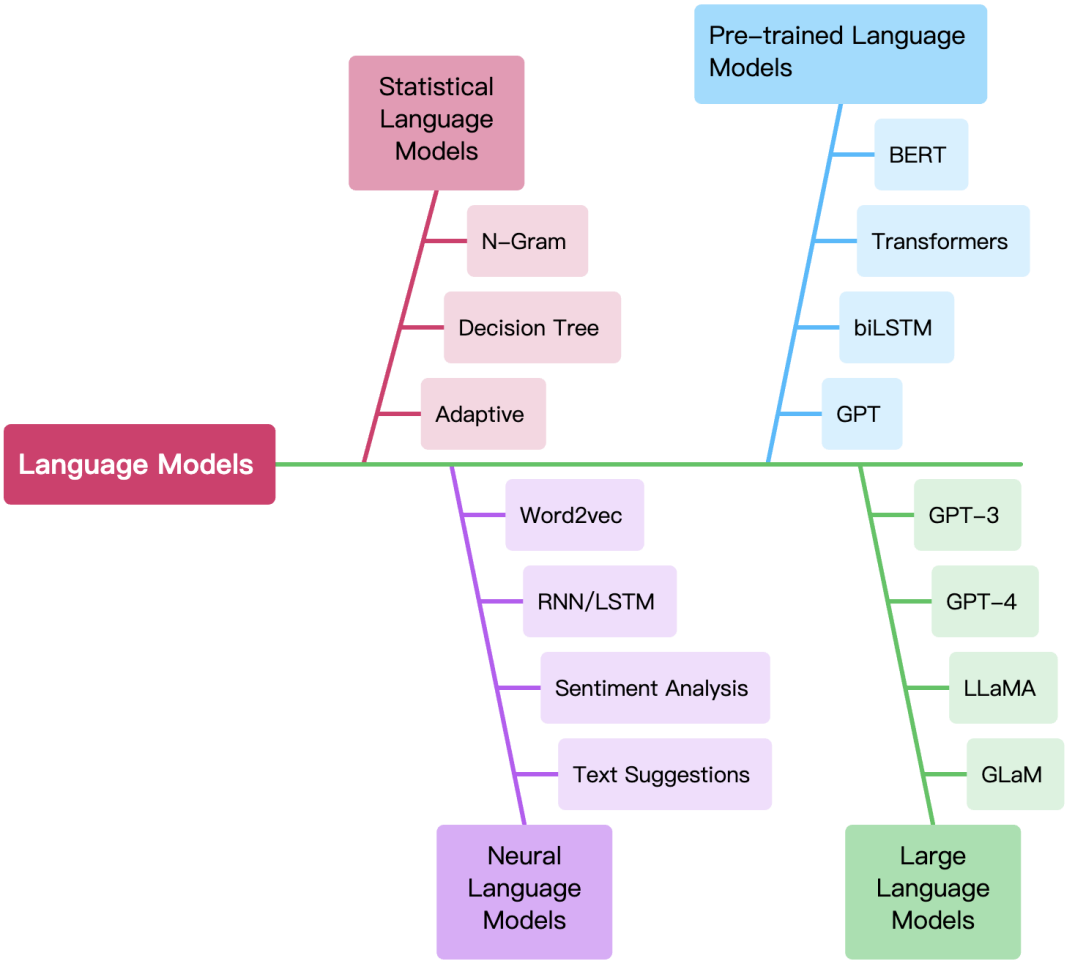

Figure 2.3: Language Models development

pre-trained language models, known as Large Language Models (LLMs)(15). LLMs are trained on vast amounts of textual data, with billions (or more) of parameters. The data comes from books, websites, and other written content, making LLMs to learn various language patterns, syntax, and semantics. Those LLMs such as GPT-3 and GPT-4 developed by OpenAI, utilizes the Transformers architecture and employs deep learning for training(16). Transformers excel at processing sequential data, making them well-suited for tasks involving text.

The Large Language Model is widely used in various fields, including in software engineering. There are many professional developers who also use LLM for code generation and code understanding(17). LLMs can generate code(18, 19) that meets specified requirements by processing structure and meaning in natural language text. For example, when

given a description such as 'Create a Python function that calculates the factorial of a number,' LLMs can generate a correct Python function to implements the desired functionality. In addition to generating code, LLMs can also understand and interpret code(20, 21). As LLMs continue to evolve, their ability to assist with more complex coding tasks will become more advanced. It helps to faster and more efficient software development.

In recent years, several large language models have been developed specifically for code-related tasks, including code summarization. For example, Code Llama – Instruct(22), released by Meta, is a special version of the Code Llama model fine-tuned to follow natural language instructions that including summarizing or explaining code. Another example is StarCoder(23) and its newer version StarCoder2(24), built by the BigCode project. These models are trained on trillions of tokens from open-source code and can work with many programming languages. They perform well in both code generation and code understanding tasks.

2.3 Prompt Engineering

2.3.1 Context of Prompt Engineering

Prompt engineering is the process of designing and refining input prompts to guide a language model to generate the desired output. The design of the input prompt largely influences the performance of LLMs, including models like GPT-4. Users of the LLMs can improve the processing power of the language model for specific tasks through prompt engineering. A prompt with a set of instructions that directs the LLMs on what kind of response is expected. The better the prompt, the more relevant, coherent, and accurate the model's output will be.

In the context of LLMs, a prompt typically have following elements (see Figure 2.4:

- **Context:** A brief description or background information that helps model to understand the task and better responses.
- **Instructions:** A clear guidance or task about what the model is supposed to do.
- **Input Data:** The specific data or text that the model needs to process.
- **Output Indicator:** The type or format of the output.

Prompt engineering is the process of refining these four components to ensure that the model produces accurate, contextually relevant and useful results.

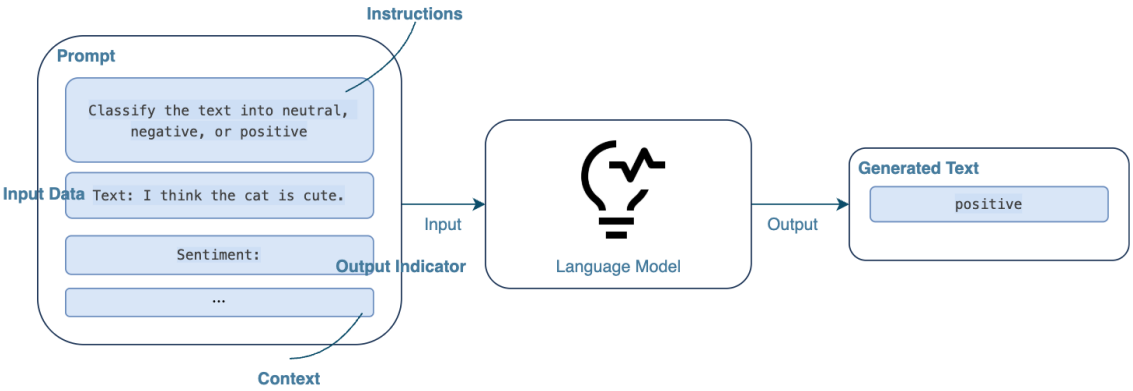

Figure 2.4: Prompt to LLM

2.3.2 Prompt engineering techniques

The concept of prompt engineering was introduced earlier, and next we will introduce different prompt engineering techniques.

- **Zero-Shot Prompting.** In zero-shot prompting, the model is given a task without any examples or demonstrations. LLMs like GPT-3, GPT-4, and Claude 3 have been trained on vast amounts of data, which makes zero-shot prompting effective(25).

- **One-Shot Prompting.** One-shot prompting is a prompt that provides a single example of the task to guide the model. One-shot prompts help the model understand the structure and expectations for the output. However, the model may not perform as well on some complex tasks as those with more examples, as it is only given as an example.

- **Few-Shot Prompting.** Few-shot prompting provides multiple examples to help the model understand the task more effectively. This approach is useful for tasks requiring consistency across multiple examples. However, increasing the number of examples also leads to an increase in the amount of tokens consumed by input, and the choice of examples can also affect the output of the model as well as create bias.

- **Chain-of-Thought (CoT) Prompting.** The three basic prompting techniques mentioned above are the first step in understanding prompt engineering. They provide foundational methods that do not require users to know more complex strategies. These techniques are easier to understand and use. On the other hand, CoT Prompting is a technique for reasoning and logic. It requires users to have a deeper

understanding of LLMs, especially for tasks that require higher levels of reasoning. According to Wei et al. (2022)(26), introduced CoT prompting that allows models to break down multi-step problems into intermediate steps. This process improves the model's reasoning abilities, allowing it to handle more complex tasks such as logical reasoning and problem-solving. Instead of providing a direct answer, CoT prompts guide the model to explain its thought process, makes the model's reasoning clearer and more accurate. In traditional prompting, the model might directly give an answer without showing any steps. Although this works well for simple tasks, more complex ones, like reasoning, problem-solving, or multi-step calculations, greatly benefit from CoT prompting. With CoT prompting, for example, the prompts would add the thought steps and final answer for a multi-step math problem, just like how humans solve problems by breaking them down into smaller, logical steps.

- **Automatic Chain-of-Thought (Auto-CoT) Prompting.** Although CoT improves the reasoning abilities of LLMs, creating high-quality CoT examples manually is time-consuming and inefficient. Auto-CoT Prompting is an advanced extension of the CoT technique. Unlike traditional CoT Prompting, where the user manually provides step-by-step instructions to guide the model through the reasoning process, Auto-CoT automates this process by adding a "Let's think step-by-step" prompt. It allows the language model to generate its own reasoning steps without needing explicit human input. It automatically identifies the logical structure needed to solve a problem, makes the reasoning process clearer and more coherent. As a result, Auto-CoT reduces the need for extensive prompt engineering, making the whole process more efficient.

- **Self-Consistency Prompting.** Self-Consistency is an advanced prompting technique used to improve the reliability and accuracy of outputs generated by LLMs. It starts by using chain-of-thought prompting to guide the model. Instead of relying on a single output, the model generates multiple possible answers(27). It then selects one response that appears most frequently or aligns most closely with the reasoning and context provided as output. This method helps LLMs produce more reliable and accurate outputs, making it an effective tool for complex tasks.

There are many other advanced techniques in prompt engineering, such as Reinforcement Learning with Human Feedback (RLHF), In-Context Learning, and Adaptive Prompting. However, these are outside the scope of this paper and will not be discussed in detail here.

3

Related Work

In this section, we will review research that is relevant to this study. Although there is limited research directly focused on using Large Language Models (LLMs) for automated GitHub repository description generation, several studies and techniques are related to this task.

3.1 Traditional Approaches to Summarization

In our study, we aim to automate the generation of GitHub repository descriptions. Automated description generation has already been the subject of numerous studies. Hellman et al. (2021)(11) investigate how developers write repository "About" descriptions on GitHub and proposes a template called LSP (Language, Software technology, Purpose) for creating clear, concise Github repository descriptions. It compares manual and natural language summarization techniques automated generating GitHub repository descriptions. The research finds that automatically generated summaries were almost as effective as the original descriptions in terms of content coverage and accuracy. This study uses the LSP template as a foundation for generating descriptions, which is also adopted in our work to guide the automatic generation of GitHub repository descriptions.

Doan et al. (2023)(28) introduces GitSum, a novel approach to fill in missing GitHub repository “About” fields by summarizing the README content. GitSum is built on pre-trained language Transformer models (using BART and T5 architectures) trained on data from existing repositories.

Both of these studies rely on traditional summarization techniques, but our approach differs by utilizing an LLM-based method. The primary reason for this is the large number of repositories in our dataset that lack README files. To address this, we use LLMs

to analyze other relevant data, such as the repository's directory structure, entry files and commit messages, ensuring that the generated descriptions are both coherent and contextually relevant.

There are also several studies focused on automated summarization for other software artifacts. Liu et al. (2019)(29) propose a sequence-to-sequence model for generating pull request (PR) descriptions from commit messages and code comments. Similarly, other research has focused on code summarization(30)(31) and bug report(32), all of which share common techniques applicable to our task.

3.2 Summarization Using LLMs

With the development of large language models, research has increasingly focused on generating summaries using LLMs. Ahmed et al. (2022)(33) explore the use of few-shot learning with large language models (LLMs) for project-specific code summarization. They find a GPT-based model Codex can generate high-quality code summaries with just a few examples. While their work focuses on function-level code summarization using LLMs, our research differs in its focus on repository-level description generation.

Bhaskar et al. (2023)(34) demonstrate the strong zero-shot summarization ability of GPT-3.5 on large collection of opinion reviews. The authors develop prompt pipelines (e.g. recursive summarization and salient-content selection) to summarize large collections of user reviews with no model training.

Unlike the above two studies that use only one prompt technique, Block et al. (2023)(35) explored various prompt strategies, including zero-shot, few-shot, and audience-specific instructions, to generate concise summaries from user interaction logs using ChatGPT. The authors introduce a recursive summarization technique, inspired by chain-of-thought prompting, to handle long logs more effectively. They evaluated the impact of these strategies on the quality of the generated summaries, focusing on their effects on factuality, accuracy, and overall coherence.

These studies highlight various techniques for using LLMs in automated summarization, ranging from few-shot and zero-shot learning to audience-specific instructions and recursive summarization. These approaches have influenced the design of prompts in our research, guiding how we structure inputs to optimize repository description generation for GitHub repositories.

4

Methodology

4.1 Overall System Architecture

To automated generation of descriptions for GitHub repositories, we developed a modular system architecture that enables large-scale data extraction, preprocessing, and language model-based text generation. The system is designed to be scalable, adaptable to various repository structures, and aligned with practical constraints such as API rate limits and the need for structured storage.

The overall workflow consists of the following core components, each of which will be discussed in detail in the subsequent sections.

1. GitHub Repository Extraction: This module utilizes the GitHub REST API to retrieve both metadata and content from all repositories within a target organization. To reduce redundancy and focus on useful context for description generation, we selectively retain fields that contain potentially descriptive or structural information from the extensive set returned by the API.

2. Data Preprocessing: The data preprocessing module is responsible for filtering, cleaning, and structuring the extracted repository data to ensure its suitability for description generation. This includes removing invalid repositories, extracting representative files, cleaning commit messages, and excluding irrelevant or noisy content such as configuration files, binaries, or non-informative commits. The result is a clean and informative dataset that serves as the input to the language model.

3. Description Generation with LLM: The cleaned repository data is then passed to a large language model—GPT-4o via Azure OpenAI API—to generate a concise, informative description for each repository that lacks one. The input prompt is carefully constructed to include both repository data and generation instructions, guiding the model to

produce relevant and context-aware outputs. The output is a brief natural language summary that outlines the repository's purpose, programming language, primary technologies, and intended users.

Throughout the entire pipeline, both intermediate and final outputs are persistently stored in structured JSON files. This includes extracted metadata, cleaned repository content, and generated descriptions. To support scalability and avoid memory overload, the data is partitioned into manageable batches (e.g., 200 repositories per file). This design not only facilitates modular debugging and error recovery at each processing stage, but also enables seamless integration with downstream components, such as the internal solution recommendation system.

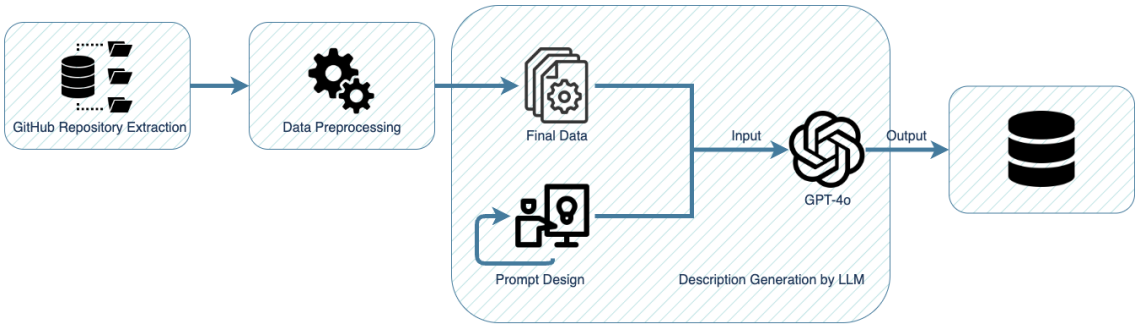

Figure 4.1: Overall System Architecture

4.2 Data Extraction

We begin the GitHub repository mining process at the dataset generation stage. Using the GitHub API(`https://api.github.com/orgs/{ORG}/repos`), we fetched basic information for 2,218 repositories within the company. From this large volume of basic information, we identified fields that could be used for generating descriptions and excluded certain fields that could not provide descriptive information. We used the GitHub API endpoints to extract the repository's file contents, commit history, and README file content from each repository.

Among all the files, README files undoubtedly contain the most descriptive information, making them highly valuable for generating descriptions. For example, the work by Hellman et al.(11) focused solely on analyzing README files. However, we found that 39.4% of repositories lack a README file. In such cases, the repository's directory structure can also provide useful information. Additionally, comments within the source code

are essential for code analysis, as they offer key insights. The amount of code in each repository varies, and to avoid excessive storage of source code, we only extract the entry files (such as main.py or index.js). We implemented a function that extracts the structure and contents of GitHub repositories. The function traverses the repository directories using a stack to ensure all paths are visited. It targets files containing specific keywords such as those for "main", "index", and "app" by checking each file's name. Files that meet these criteria are downloaded and stored in the dataset, while the structure of the repository is maintained.

At the data extraction stage, an important issue is the GitHub API's rate limit. Each time we go one level deeper into a directory and extract a file, an API request is made. All of these requests count towards the personal rate limit of 5,000 requests per hour.To address this, we implemented a strategy that optimizes requests. By checking the remaining request quota before making any API calls, we ensure that the system pauses when approaching the rate limit. If the number of remaining requests falls below a defined threshold, the process waits until the rate limit resets. This approach minimizes the risk of exceeding the request limits while maintaining data extraction efficiency. Additionally, we added a buffer to handle multiple requests in a loop, allowing continuous data fetching without interruptions.

The extracted dataset includes the fields shown in the Table 4.1.

Table 4.1: Extracted Dataset Fields

Field Name	**Field Description**
name	Repository name
size	Size of the repository
html_url	URL to the repository on GitHub
description	User provide Description of the repository (if available)
language	Primary programming languages used in the repository
topics	User selected topics related to the repository
main_files	List of files matched as entry file
commit_messages	List of commit messages in the repository
structure	List of path refer to repository directory structure
stargazers_count	Number of stars received by the repository
forks_count	Number of forks of the repository
open_issues_count	Number of open issues in the repository

4.3 Data Preprocessing

After extracting the repository dataset containing the aforementioned fields, we perform data preprocessing to ensure the data is clean and structured, making it suitable for the LLM to generate descriptions. We preprocessed the extracted repository according to the following processes:

Filtering repositories: We focus on ensuring the repository data is accurate and usable. We remove invalid repositories by skipping those that are empty ($size = 0$) or have no code (*language* is empty). Out of the 2,218 repositories, 49 are empty, and 78 have no code. We add a new field *has_description* to mark repositories with or without descriptions. We marked repositories without descriptions, which make up 49.8%. If a repository already has a description, it is retained. This process helps ensure that only relevant and complete data is passed on for further processing and description generation.

Preprocessing Repository Files: We process the repository's file data to ensure that only representative and relevant files are used by the LLM. We filter entry files by using regular expressions to match filenames that represent the core purpose of the repository (e.g., main.*, index.*, app.* like main.py, index.html, app.js). We avoid partial string matches to prevent errors, such as matching "appointment.js." Irrelevant files, including configuration files (.config, .yaml, .json), log files (.log, .tmp), and image files (e.g., .png, .jpg), are excluded to ensure only text-based files are processed.

Filtering commit messages: We extract the most recent 20 commit messages from each repository to avoid excessive data. We filter out non-informative commits (e.g., "Update README" or "Fix bug") to ensure that only meaningful commit messages, which provide insights into the repository's changes, are included for further analysis.

The whole preprocessing procedure shown in Figure 3. This preprocessing ensures that the data is structured and clean, enabling the LLM to generate accurate and useful descriptions.

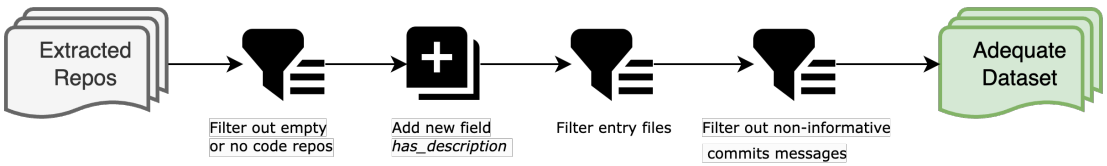

Figure 4.2: Data Preprocessing Procedure

4.4 Description Generation with LLM

4.4.1 LLM Selection

Before starting the process of LLM-based description generation, we need to select an appropriate large language model to perform this task. We selected GPT-4o(36) due to its high performance and exceptional ability to generate high-quality, contextually relevant text and handle complex NLP tasks. GPT-4o is particularly effective at understanding software-related content and has demonstrated fast response times in generating coherent and meaningful outputs. In the research from independent third party lab METR(37), GPT-4o's performance was compared to that of other models, including Claude 3 and Claude 3.5, across a range of task completion scenarios with varying time limits. The results indicated that GPT-4o consistently outperformed the other models in terms of the fraction of tasks completed, even under tight time constraints, demonstrating its efficiency and effectiveness in generating accurate outputs. The model's ability to maintain high performance, especially when compared to human performance under similar time restrictions, further confirms its suitability for automated description generation tasks. Additionally, its integration with the Azure OpenAI API(38) provides robust enterprise-level support, including enhanced security and the reliability of Azure's enterprise commitments. These features make GPT-4o an ideal choice for generating GitHub repository descriptions efficiently and accurately.

While there are also large language models specifically designed for code summarization, such as Code LLaMA and StarCoder, these models are typically trained for function-level summarization, where the input is a single code block or method. However, the goal of this study is to generate descriptions at the repository level, using metadata, directory structure, and selected entry files. This requires understanding the broader context of a project, rather than summarizing a single function. Although DeepSeek-Coder(39) is trained on repository-level data and performs well in code-related tasks, its performance in code generation is slightly better than GPT-3 but still behind GPT-4o. In addition, many of these specialized models are not easily accessible or deployable in enterprise environments. For these reasons, they were not used in this study.

4.4.2 Criteria for High-Quality Repository Descriptions

To generate meaningful repository descriptions, it is essential to define what constitutes a useful and high-quality repository description. In many open-source repositories, some de-

scriptions are clear and detailed, especially in well-maintained projects such as Pacco(40), brpc(41), and cpprestsdk(42). Others are short, vague, or missing altogether. This variation makes it difficult to follow a universal standard. Hellman et al. (2021)(11) proposed the LSP (Language, Software Technology, Purpose) template as a guideline to help developers write understandable and comprehensive descriptions. In our work, we adopted the LSP structure as the baseline for describing software repositories. The decision to use and adapt this structure was based on feedback from internal experts. During the early stage of the project, we asked several technical staff members to each select five descriptions from internal repositories that they personally found useful. Most of the selected examples followed the LSP (Language, Software technology, Purpose) structure. In addition, many of these examples mentioned the intended users or internal teams, such as "used by the automation pipeline" or "designed for frontend developers." Based on this observation, we extended it with a fourth element: the target user. In an enterprise setting, projects are often built for specific teams, systems, or workflows. Internal repositories may also involve domain-specific terminology or abbreviations that are only meaningful to certain users. By explicitly including the intended user group in the description, we help developers quickly understand whether a project is relevant to their role or domain. This addition makes it easier to navigate large, diverse codebases. This extended structure became the basis for our prompt design.

4.4.3 Prompt Design

The prompt design in this work is based on the description structure defined in Section 4.4.2, which combines the LSP (Language, Software Technology, Purpose) framework proposed by Hellman et al. (2021)(11) with an additional target user dimension. Based on this extended template, we selected repository descriptions that conform to the LSP structure to serve as few-shot examples in our prompt design.

To guide the language model in generating consistent and high-quality outputs, we designed a structured few-shot prompt. It was based on the extended LSP template and included clear instructions. The selected examples show how to describe a repository by mentioning its programming language, key technologies, purpose, and target users. We provided two real-world examples to help the model understand both the format and the tone. These examples are concise, natural, and technically informative. They help the model generate descriptions similar to those written by developers in real-world settings.

We also tested zero-shot prompts. These prompts only include the task instruction and repository metadata, without any examples. While they are shorter and more flexible,

they often produce generic or incomplete outputs. For this reason, we found that zero-shot prompting was not suitable for our task.

The final prompt includes an expected output format, writing guidelines, and examples. It ends with a new input containing only repository metadata. The model is expected to follow the same structure and tone as in the examples. This design helps improve both the quality and consistency of the generated descriptions.

Table 4.2: Prompt Variants Used for Description Generation

ID	Type	Structure Overview	Generate Output
P1	Zero-shot	Task description + repository metadata	Too long, generic or incomplete
P2	Few-shot	Examples + new repository metadata	Not always contain the information we want
P3	Few-shot(LSP)	LSP with user format structure + same as P2	Good structure and coverage
P4	Few-shot (LSP) + instructions	Same as P3 + instructions	Concise natural and readable

These prompt variants were used to explore how structure, examples, and task framing influence the quality of LLM-generated descriptions. The most effective prompt (P4) was selected for the final evaluation. A simplified version of the prompt (with omitted examples) is shown in Figure 4.3.

4.4.4 Repository Description Generation

To automatically generate human-readable descriptions for repositories, we employ a large language model (LLM), specifically OpenAI's GPT-4o, in combination with a structured prompt. The generation process is designed to be scalable and token-efficient, enabling the model to handle hundreds of repositories without exceeding input limits.

We employ OpenAI's GPT-4o model via its chat.completions.create endpoint. Each prompt sent to the LLM consists of two parts: (1) an instruction, which corresponds to the final prompt variant (P4) described in Section 4.4.2, and (2) a batch of repository data represented in JSON format. The construction and preprocessing of this data are described in Section 4.2. The model receives both components as user messages in the chat format.

Table 4.3 summarizes the configuration parameters used for model calls.

Generate a informative description for given GitHub repositories in JSON format.
Description provides those information:

- Language(Specify the primary programming language used in the repository)
- Software Technology(List the main frameworks or libraries employed)
- Purpose(Explaining what this tool does and why it's useful)
- Target User(If applicable, indicate Who this tool is for (e.g., civil3D users, ESRI users). If this information is not available, omit this part.)

Guidelines:
1. If a README file is provided, prioritize analyze its content. If the README is missing or lacks sufficient details, rely on the given other data.
2. Ensure the final description remains structured and concise, do not include redundant information, even when extracting information from the README.
3. Keep the description natural and readable, as if written by a developer, not a machine.
4. Here is a good example description for another repository: {description 1}
5. Here is another example description: {description 2}

Return output is strict JSON format only include name and description.

Figure 4.3: Final prompt used for description generation.

The temperature parameter controls the degree of randomness in output generation. Higher values produce more diverse and creative outputs, while lower values lead to more focused and fact-based responses. In this study, we fix the temperature to 0 to ensure that the same input always results in the same output. Repositories are processed in separate batches. Using a fixed temperature ensures that the model's output does not change due to differences in batch content or order. This aligns with our goal of generating precise and informative technical descriptions based on structured inputs.

Due to the token limits imposed by the API, a dynamic batching strategy was implemented to maximize efficiency while staying within constraints. To manage input size and maximize efficiency, we implement a token-aware batching strategy. The GPT-4o API allows a maximum context length of 128,000 tokens per request. To reduce the risk of exceeding this limit, we conservatively cap each request at 100,000 tokens. Within this budget, 324 tokens are reserved for the instruction prompt, and 50 tokens are allocated for each expected description.

Table 4.3: LLM Configuration Parameters

Parameter	**Value**
Model	gpt-4o
API Version	2024-06-01
Temperature	0
Encoding	o200k_base
Max Input Tokens	100,000
Expected Generation Tokens	50 per repository

Before generation, the token count of each repository's JSON-formatted metadata is estimated using OpenAI's official tokenizer (tiktoken). This ensures accurate alignment with the model's internal encoding. Repositories are then grouped into batches such that the total estimated token count remains under the cap. The batch size is adjusted dynamically based on the actual token size of each repository. This design enables robust and scalable generation, while avoiding errors caused by token overflow.

5

Evaluation

This section presents the evaluation of the proposed approach, focusing on how the research questions introduced in Section 1 are addressed in practice.

RQ1 (*How to extract and refine metadata fields to effectively capture the essential characteristics of a repository?*) is addressed through the design and implementation of a metadata extraction and preprocessing pipeline, which focuses not only on data retrieval but also on selecting meaningful subsets of information for summarization. Since this aspect is not the main subject of evaluation, a full explanation of the extraction strategy is discussed in earlier sections and summarized in the conclusion.

RQ2 (*How to automatically generate precise and useful repository descriptions by LLM?*) is addressed through a two-part solution: a prompt-based generation strategy and a quantitative evaluation framework. This evaluation aims to to determine whether the descriptions generated by a large language model (LLM) accurately and meaningfully reflect the content and intent of the source repositories, thus addressing RQ2.

5.1 Evaluation Metrics

We adopt BERTScore(43) as the evaluation metric for this study. BERTScore is a recent and widely used metric designed to measure the similarity between two texts based on contextual embeddings from large pre-trained transformer models such as BERT, RoBERTa, or DeBERTa(44). Instead of comparing exact n-gram matches as in traditional metrics like ROUGE or BLEU, BERTScore computes a similarity score between each token in the generated (candidate) text and the reference text using their vector representations(45). These token embeddings are derived from deep layers of the transformer model and capture rich semantic information. Compared to n-gram overlap metrics like ROUGE, BERTScore

is less sensitive to summary length(46) and has a stronger correlation with human assessments of relevance and coherence(47), making it particularly well-suited for our task.

BERTScore ranging from 0 to 100, where higher values indicate better quality. The final BERTScore consists of three components, precision and recall measure the quality of the match between the generated text and the reference text, respectively, while the F1 score is their reconciled mean.

5.2 Experiment Settings

The evaluation was conducted on a dataset of 1,100 internal GitHub repositories collected from company codebase. Each repository includes a range of metadata fields, such as the repository name, primary programming language, list of topics, directory structure, main files, recent commit messages, README content, the original human-written description, and the description generated by the LLM.

To perform a quantitative evaluation, we randomly sampled 30% of the dataset, resulting in a subset of 330 repositories. This subset was used for computing similarity between the LLM-generated descriptions and reference content acquired from repository metadata.

For each repository, a reference text was constructed by concatenating all metadata fields except the original description and the LLM-generated description. This design ensures that the reference captures the content the LLM had access to during generation, allowing us to assess how well the generated output semantically reflects the input. The LLM-generated description was treated as the candidate in the evaluation. We employed BERTScore to measure the semantic similarity between candidate and reference texts, using the pre-trained *microsoft/deberta-large-mnli* model. BERTScore providing precision, recall, and F1 scores.

To avoid memory constraints during computation, the evaluation was performed in batches of 32 repositories on a GPU. This batching strategy also makes the evaluation pipeline scalable to larger datasets.

In addition to the LLM-generated descriptions, we include two simple baselines for comparison. The first is a random baseline, in which each repository is assigned a description randomly selected from another repository in the dataset. This serves as a lower bound and helps verify whether the generated descriptions are meaningfully aligned with the corresponding repositories.

The second is a rule-based baseline, which constructs a description by concatenating fixed metadata fields, such as the first 500 characters of the README file (if available) or recent

commit messages. While simple, this method can still produce reasonable summaries for well-structured repositories. It allows us to assess whether the LLM offers clear advantages over fixed-format, rule-based descriptions. All descriptions, including those from baselines, are evaluated using the same reference construction and BERTScore computation described above.

As a supplementary analysis, we compare the LLM-generated descriptions against the original human-written descriptions. This provides insight into how closely the generated output resembles human summaries in terms of fluency and semantic content.

5.3 Results and Analysis

5.3.1 Main evaluation

We report the evaluation results using BERTScore, comparing the LLM-generated descriptions with two baselines (random and rule-based).

Table 5.1 summarizes the average precision, recall, and F1 scores for all methods.

Table 5.1: BERTScore Comparison Between Description Methods and Metadata Input

Method	Precision	Recall	F1
LLM vs Metadata Input	**0.5664**	0.3789	**0.4499**
Rule-Based Baseline	0.4480	0.3600	0.3972
Random Baseline	0.4424	0.3438	0.3863

It is important to note that the difference between the input and output in our setting. The input to the LLM is not full natural language text. Instead, it consists of structured metadata and selected filenames. The output is a short description that summarizes the purpose and function of the repository. Because of this, the input and output do not have a one-to-one match in wording.

BERTScore compares the meaning of two texts. It works best when both texts are natural language sentences. In our case, the generated descriptions are not meant to repeat exact words from the input. Instead, they should express the key ideas in a clear way. So even if the BERTScore is not very high, it still shows that the model captured useful information. This makes BERTScore a helpful, although cautious, way to measure how well the output aligns with the expected description.

As shown in Table 5.1, the LLM-generated descriptions achieve the highest average F1 score (0.4499) when compared to the original repository metadata, outperforming both the

rule-based (0.3972) and random (0.3863) baselines. This suggests that the LLM is able to produce summaries that more closely reflect the semantic content of the repository inputs.

The rule-based baseline performs marginally better than the random baseline across all metrics, but both remain notably below the LLM-generated output. This gap highlights the LLM's ability to synthesize contextually meaningful descriptions, rather than simply rephrasing or combining fixed fields.

Overall, these results suggest that the LLM-based method provides a moderate improvement in semantic alignment over basic rule-based or random approaches. While the gains are not dramatic, they indicate that prompt-based generation offers added value in scenarios where high-level summaries are desired but manual descriptions are missing or incomplete.

5.3.2 Comparison Against Human-Written Descriptions

As a supplementary analysis, we evaluate the similarity between the LLM-generated descriptions and the original human-written descriptions available in the dataset. This comparison aims to assess the linguistic and semantic resemblance between the generated output and the way developers typically describe their projects.

It is important to note that these original descriptions are not treated as gold-standard references. They were directly extracted from the repositories without any quality validation, and may vary in completeness, style, and accuracy. As such, this evaluation does not assume that the human-written descriptions are objectively correct. Instead, we use them as imperfect but realistic reference points to examine whether the generated descriptions resemble natural developer-authored summaries.

We apply BERTScore in the same manner as in the main evaluation, using the original description as the reference and the LLM-generated description as the candidate. The results are presented in Table 5.2.

Table 5.2: BERTScore Between LLM-Generated and Human-Written Descriptions

Comparison	**Precision**	**Recall**	**F1**
LLM vs Human Description	0.5195	0.6257	**0.5655**

The LLM-generated descriptions achieve an average F1 score of 0.5655 in this comparison. The relatively high recall (0.6257) suggests that the generated outputs often cover key ideas present in the original descriptions. The slightly lower precision (0.5195) may indicate the inclusion of general or inferred content not explicitly stated by developers.

While this evaluation is not part of the core assessment, it provides useful insight into the human-likeness and naturalness of the generated outputs. These findings suggest that the LLM is capable of producing developer-friendly summaries that align stylistically and semantically with real-world descriptions—even in the absence of direct supervision or reference to the original text.

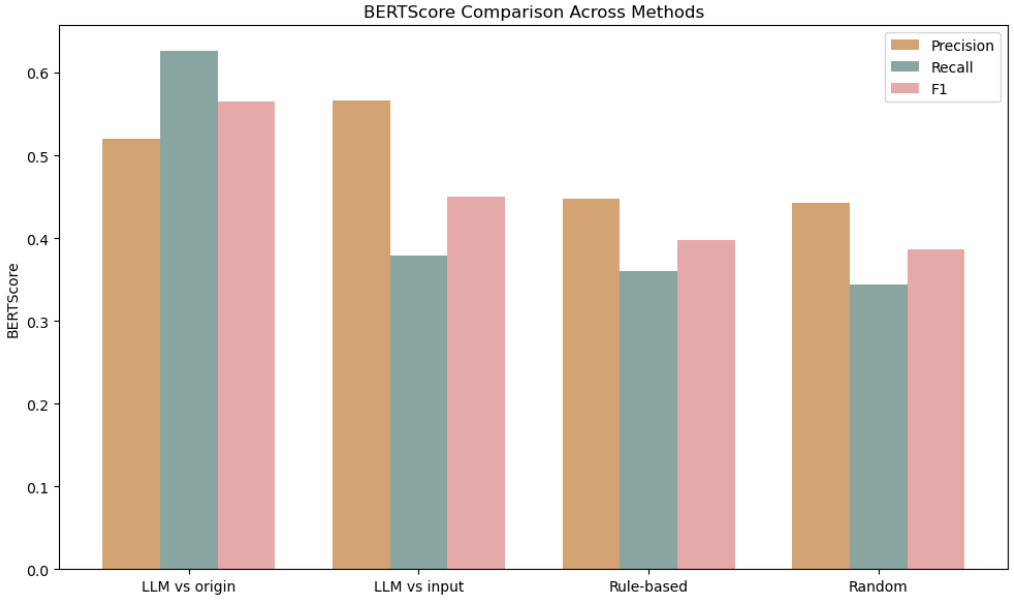

Figure 5.1: Comparison of BERTScore

Figure 5.1 provides a visual comparison of BERTScore precision, recall, and F1 across all evaluated methods.

6

Discussion

This section reflects on the implications of the experimental results in the context of the research questions defined in this study. The discussion is structured around two primary objectives: the feasibility of using large language models (LLMs) for generating repository-level descriptions and the effectiveness of such descriptions in relation to structured metadata and human-written references.

RQ1 – How to extract and refine metadata fields to effectively capture the essential characteristics of a repository? Although not the focus of the evaluation section, the solution to RQ1 underpins the entire system. A pipeline was developed to automatically retrieve structured metadata from GitHub repositories. This extraction pipeline handled batching and API rate limits to support large-scale data processing. The resulting metadata formed the foundation for LLM prompting and evaluation.

The successful application of this pipeline across 2218 repositories shows that automated metadata collection can scale to real-world datasets with moderate engineering effort. While the quality of raw metadata (e.g., README completeness, commit clarity) remains variable, the design choices ensured that the LLM had access to meaningful and diverse input information.

RQ2 – How to automatically generate precise and useful repository descriptions by LLM? This research explored a prompt-based approach using GPT-4o to generate repository-level descriptions from structured metadata. BERTScore results indicate that LLM-generated descriptions provide stronger semantic alignment with input metadata than the two baselines.

Although the improvements are moderate, the LLM demonstrates a consistent ability to synthesize relevant information into readable summaries. The comparison with human-written descriptions further suggests that the generated outputs share stylistic and

semantic similarities with real-world developer documentation. These findings indicate that LLMs can be integrated into internal developer tools to assist with documentation, indexing, or summarization of code repositories. Especially in large organizations with hundreds of under-documented repositories, LLMs could provide first-pass summaries to support onboarding, knowledge sharing, or recommendation systems.

This study did not include human evaluation, partly due to time constraints and partly because the content often requires domain-specific knowledge. Many repository descriptions involve internal tools, frameworks, or workflows that are difficult to assess accurately without deep project context. As a result, it would be unreliable to evaluate the outputs using only one or two developers without broader consensus or annotated guidelines. Furthermore, some repositories lacked meaningful metadata (e.g., empty READMEs or minimal commit messages), which may have constrained the model's ability to produce informative outputs. Several directions remain open for future research. First, integrating human-in-the-loop feedback could improve both description quality and evaluation reliability. Second, prompt optimization techniques such as few-shot learning with curated examples or retrieval-augmented generation may enhance alignment with user expectations. Lastly, expanding the evaluation framework to include additional metrics (e.g., factual consistency, readability, usefulness) would provide a more comprehensive understanding of output quality.

7

Conclusion

Our work presents an automated system for generating descriptive summaries of GitHub repositories using large language models (LLMs). By combining structured metadata extraction, prompt-based LLM generation, and semantic evaluation, the system addresses the widespread issue of missing or unclear repository descriptions in large organizations. Firstly, this work develops a scalable data extraction and preprocessing pipeline tailored for LLM input, enabling efficient retrieval and structuring of repository metadata. Secondly, it designs and evaluates prompt strategies based on the LSP (Language, Software, Purpose) framework, extended with user information to enhance description relevance. Finally, it introduces a quantitative assessment using BERTScore to measure the semantic alignment between the generated descriptions, input metadata, and human-written summaries. The results show that LLMs can reliably synthesize repository information into readable summaries. This work suggests practical applications for internal documentation, indexing, and recommendation systems. Future research could explore human-in-the-loop refinement, richer evaluation metrics, and enhanced prompt designs to further improve output quality and usefulness.

References

[1] **Build software better, together — github.com**. `https://github.com/about`. 1

[2] Yang Zhang, Hanlei Jin, Dan Meng, Jun Wang, and Jinghua Tan. **A comprehensive survey on process-oriented automatic text summarization with exploration of llm-based methods**. *arXiv preprint arXiv:2403.02901*, 2024. 1

[3] Junxiao Han, Shuiguang Deng, Xin Xia, Dongjing Wang, and Jianwei Yin. **Characterization and Prediction of Popular Projects on GitHub**. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, **1**, pages 21–26, 2019. 2

[4] JaeWon Kim, JeongA Wi, and YoungBin Kim. **Sequential recommendations on github repository**. *Applied Sciences*, **11**(4):1585, 2021. 2

[5] Huajie Shao, Dachun Sun, Jiahao Wu, Zecheng Zhang, Aston Zhang, Shuochao Yao, Shengzhong Liu, Tianshi Wang, Chao Zhang, and Tarek Abdelzaher. **paper2repo: GitHub Repository Recommendation for Academic Papers**. In *Proceedings of The Web Conference 2020*, WWW '20, page 629–639, New York, NY, USA, 2020. Association for Computing Machinery. 2

[6] Yun Zhang, David Lo, Pavneet Singh Kochhar, Xin Xia, Quanlai Li, and Jianling Sun. **Detecting similar repositories on GitHub**. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 13–23, 2017. 2

[7] Yiren Zhou, Yu Qiao, and Tao Xu. **Exploring the Characteristics of Popular Deep Learning GitHub Repositories**. In *2023 International Conference on Intelligent Computing and Next Generation Networks (ICNGN)*, pages 1–6. IEEE, 2023. 2

[8] Guoming Long and Tao Chen. **On Reporting Performance and Accuracy Bugs for Deep Learning Frameworks: An Exploratory Study from GitHub**. In *Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*, EASE '22, page 90–99, New York, NY, USA, 2022. Association for Computing Machinery. 2

[9] Dhivyabharathi Ramasamy, Cristina Sarasua, Alberto Bacchelli, and Abraham Bernstein. **Workflow analysis of data science code in public GitHub repositories**. *Empirical Software Engineering*, **28**(1):7, 2023. 2

[10] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Jordan Henkel, Matteo Interlandi, Subru Krishnan, Brian Kroth, Venkatesh Emani, Wentao Wu, Ce Zhang, Markus Weimer, Avrilia Floratou, Carlo Curino, and Konstantinos Karanasos. **Data Science Through the Looking Glass: Analysis of Millions of GitHub Notebooks and ML.NET Pipelines**. *SIGMOD Rec.*, **51**(2):30–37, July 2022. 2

[11] Jazlyn Hellman, Eunbee Jang, Christoph Treude, Chenzhun Huang, and Jin L. C. Guo. **Generating GitHub Repository Descriptions: A Comparison of Manual and Automated Approaches**, 2021. 2, 13, 16, 20

[12] Nazatul Nurlisa Zolkifli, Amir Ngah, and Aziz Deraman. **Version control system: A review**. *Procedia Computer Science*, **135**:408–415, 2018. 5

[13] Matti Vuorre and James P Curley. **Curating research assets: A tutorial on the Git version control system**. *Advances in Methods and Practices in Psychological Science*, **1**(2):219–236, 2018. 5

[14] Zichong Wang, Zhibo Chu, Thang Viet Doan, Shiwen Ni, Min Yang, and Wenbin Zhang. **History, development, and principles of large language models: an introductory survey**. *AI and Ethics*, pages 1–17, 2024. 7

[15] Muhammad Usman Hadi, Rizwan Qureshi, Abbas Shah, Muhammad Irfan, Anas Zafar, Muhammad Bilal Shaikh, Naveed Akhtar, Jia Wu, Seyedali Mirjalili, et al. **A survey on large language models: Applications, challenges, limitations, and practical usage**. *Authorea Preprints*, **3**, 2023. 8

[16] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam,

Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. **Language Models are Few-Shot Learners**. *CoRR*, **abs/2005.14165**, 2020. 8

[17] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. **Large Language Models for Software Engineering: A Systematic Literature Review**, 2024. 8

[18] Youjia Li, Jianjun Shi, and Zheng Zhang. **An Approach for Rapid Source Code Development Based on ChatGPT and Prompt Engineering**. *IEEE Access*, **12**:53074–53087, 2024. 8

[19] Paula Maddigan and Teo Susnjak. **Chat2VIS: Generating Data Visualizations via Natural Language Using ChatGPT, Codex and GPT-3 Large Language Models**. *IEEE Access*, **11**:45181–45193, 2023. 8

[20] Man-Fai Wong, Shangxin Guo, Ching-Nam Hang, Siu-Wai Ho, and Chee-Wei Tan. **Natural Language Generation and Understanding of Big Code for AI-Assisted Programming: A Review**. *Entropy*, **25**(6):888, June 2023. 9

[21] Paheli Bhattacharya, Manojit Chakraborty, Kartheek N S N Palepu, Vikas Pandey, Ishan Dindorkar, Rakesh Rajpurohit, and Rishabh Gupta. **Exploring Large Language Models for Code Explanation**, 2023. 9

[22] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. **Code Llama: Open Foundation Models for Code**, 2024. 9

[23] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li,

Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. **StarCoder: may the source be with you!**, 2023. 9

[24] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauss, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. **StarCoder 2 and The Stack v2: The Next Generation**, 2024. 9

[25] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. **A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications**, 2025. 10

[26] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. **Chain of Thought Prompting Elicits Reasoning in Large Language Models**. *CoRR*, **abs/2201.11903**, 2022. 11

[27] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. **Self-Consistency Improves Chain of Thought Reasoning in Language Models**, 2023. 11

[28] Thu T. H. Doan, Phuong T. Nguyen, Juri Di Rocco, and Davide Di Ruscio. **Too long; didn't read: Automatic summarization of GitHub README.MD with Transformers**. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, EASE '23, page 267–272, New York, NY, USA, 2023. Association for Computing Machinery. 13

[29] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. **Automatic Generation of Pull Request Descriptions**. *CoRR*, **abs/1909.06987**, 2019. 14

[30] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. **A Transformer-based Approach for Source Code Summarization**. *CoRR*, **abs/2005.00653**, 2020. 14

[31] Ensheng Shi, Yanlin Wang, Lun Du, Junjie Chen, Shi Han, Hongyu Zhang, Dongmei Zhang, and Hongbin Sun. **Neural Code Summarization: How Far Are We?** *CoRR*, **abs/2107.07112**, 2021. 14

[32] Songqiang Chen, Xiaoyuan Xie, Bangguo Yin, Yuanxiang Ji, Lin Chen, and Baowen Xu. **Stay Professional and Efficient: Automatically Generate Titles for Your Bug Reports**. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 385–397, 2020. 14

[33] Toufique Ahmed and Premkumar Devanbu. **Few-shot training LLMs for project-specific code-summarization**, 2022. 14

[34] Adithya Bhaskar, Alex Fabbri, and Greg Durrett. **Prompted Opinion Summarization with GPT-3.5**. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Findings of the Association for Computational Linguistics: ACL 2023*, pages 9282–9300, Toronto, Canada, July 2023. Association for Computational Linguistics. 14

[35] Jeremy Block, Yu-Peng Chen, Abhilash Budharapu, Lisa Anthony, and Bonnie Dorr. **Summary Cycles: Exploring the Impact of Prompt Engineering on Large Language Models' Interaction with Interaction Log Information**. In Daniel Deutsch, Rotem Dror, Steffen Eger, Yang Gao, Christoph Leiter, Juri Opitz, and Andreas Rücklé, editors, *Proceedings of the 4th Workshop on Evaluation and Comparison of NLP Systems*, pages 85–99, Bali, Indonesia, November 2023. Association for Computational Linguistics. 14

[36] OpenAI, :, Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, Aleksander Mądry, Alex Baker-Whitcomb, Alex Beutel, Alex Borzunov, Alex Carney, Alex Chow, Alex Kirillov, Alex Nichol, Alex Paino, Alex Renzin, Alex Tachard Passos, Alexander Kirillov, Alexi Christakis, Alexis Conneau, Ali Kamali, Allan Jabri, Allison Moyer, Allison Tam, Amadou Crookes, Amin Tootoochian, Amin Tootoonchian, Ananya Kumar, Andrea Vallone, Andrej Karpathy, Andrew Braunstein, Andrew Cann, Andrew Codispoti, Andrew Galu, Andrew Kondrich, Andrew Tulloch, Andrey Mishchenko, Angela Baek, Angela Jiang, Antoine Pelisse, Antonia Woodford, Anuj Gosalia, Arka Dhar, Ashley Pantuliano, Avi Nayak, Avital Oliver, Barret Zoph, Behrooz Ghorbani, Ben Leimberger, Ben Rossen, Ben Sokolowsky, Ben Wang, Benjamin Zweig, Beth Hoover, Blake Samic, Bob McGrew, Bobby Spero, Bogo Giertler, Bowen Cheng, Brad Lightcap, Brandon Walkin, Brendan Quinn, Brian Guarraci, Brian Hsu, Bright Kellogg, Brydon Eastman, Camillo Lugaresi, Carroll Wainwright, Cary Bassin, Cary Hudson, Casey Chu, Chad Nelson, Chak Li, Chan Jun Shern, Channing Conger, Charlotte Barette, Chelsea Voss, Chen Ding, Cheng Lu, Chong Zhang, Chris Beaumont, Chris Hallacy, Chris Koch, Christian Gibson, Christina Kim, Christine Choi, Christine McLeavey, Christopher Hesse, Claudia Fischer, Clemens Winter, Coley Czarnecki, Colin Jarvis, Colin Wei, Constantin Koumouzelis, Dane Sherburn, Daniel Kappler, Daniel Levin, Daniel Levy, David Carr, David Farhi, David Mely, David Robinson, David Sasaki, Denny Jin, Dev Valladares, Dimitris Tsipras, Doug Li, Duc Phong Nguyen, Duncan Findlay, Edede Oiwoh, Edmund Wong, Ehsan Asdar, Elizabeth Proehl, Elizabeth Yang,

Eric Antonow, Eric Kramer, Eric Peterson, Eric Sigler, Eric Wallace, Eugene Brevdo, Evan Mays, Farzad Khorasani, Felipe Petroski Such, Filippo Raso, Francis Zhang, Fred von Lohmann, Freddie Sulit, Gabriel Goh, Gene Oden, Geoff Salmon, Giulio Starace, Greg Brockman, Hadi Salman, Haiming Bao, Haitang Hu, Hannah Wong, Haoyu Wang, Heather Schmidt, Heather Whitney, Heewoo Jun, Hendrik Kirchner, Henrique Ponde de Oliveira Pinto, Hongyu Ren, Huiwen Chang, Hyung Won Chung, Ian Kivlichan, Ian O'Connell, Ian O'Connell, Ian Osband, Ian Silber, Ian Sohl, Ibrahim Okuyucu, Ikai Lan, Ilya Kostrikov, Ilya Sutskever, Ingmar Kanitscheider, Ishaan Gulrajani, Jacob Coxon, Jacob Menick, Jakub Pachocki, James Aung, James Betker, James Crooks, James Lennon, Jamie Kiros, Jan Leike, Jane Park, Jason Kwon, Jason Phang, Jason Teplitz, Jason Wei, Jason Wolfe, Jay Chen, Jeff Harris, Jenia Varavva, Jessica Gan Lee, Jessica Shieh, Ji Lin, Jiahui Yu, Jiayi Weng, Jie Tang, Jieqi Yu, Joanne Jang, Joaquin Quinonero Candela, Joe Beutler, Joe Landers, Joel Parish, Johannes Heidecke, John Schulman, Jonathan Lachman, Jonathan McKay, Jonathan Uesato, Jonathan Ward, Jong Wook Kim, Joost Huizinga, Jordan Sitkin, Jos Kraaijeveld, Josh Gross, Josh Kaplan, Josh Snyder, Joshua Achiam, Joy Jiao, Joyce Lee, Juntang Zhuang, Justyn Harriman, Kai Fricke, Kai Hayashi, Karan Singhal, Katy Shi, Kavin Karthik, Kayla Wood, Kendra Rimbach, Kenny Hsu, Kenny Nguyen, Keren Gu-Lemberg, Kevin Button, Kevin Liu, Kiel Howe, Krithika Muthukumar, Kyle Luther, Lama Ahmad, Larry Kai, Lauren Itow, Lauren Workman, Leher Pathak, Leo Chen, Li Jing, Lia Guy, Liam Fedus, Liang Zhou, Lien Mamitsuka, Lilian Weng, Lindsay McCallum, Lindsey Held, Long Ouyang, Louis Feuvrier, Lu Zhang, Lukas Kondraciuk, Lukasz Kaiser, Luke Hewitt, Luke Metz, Lyric Doshi, Mada Aflak, Maddie Simens, Madelaine Boyd, Madeleine Thompson, Marat Dukhan, Mark Chen, Mark Gray, Mark Hudnall, Marvin Zhang, Marwan Aljubeh, Mateusz Litwin, Matthew Zeng, Max Johnson, Maya Shetty, Mayank Gupta, Meghan Shah, Mehmet Yatbaz, Meng Jia Yang, Mengchao Zhong, Mia Glaese, Mianna Chen, Michael Janner, Michael Lampe, Michael Petrov, Michael Wu, Michele Wang, Michelle Fradin, Michelle Pokrass, Miguel Castro, Miguel Oom Temudo de Castro, Mikhail Pavlov, Miles Brundage, Miles Wang,

Minal Khan, Mira Murati, Mo Bavarian, Molly Lin, Murat Yesildal, Nacho Soto, Natalia Gimelshein, Natalie Cone, Natalie Staudacher, Natalie Summers, Natan LaFontaine, Neil Chowdhury, Nick Ryder, Nick Stathas, Nick Turley, Nik Tezak, Niko Felix, Nithanth Kudige, Nitish Keskar, Noah Deutsch, Noel Bundick, Nora Puckett, Ofir Nachum, Ola Okelola, Oleg Boiko, Oleg Murk, Oliver Jaffe, Olivia Watkins, Olivier Godement, Owen Campbell-Moore, Patrick Chao, Paul McMillan, Pavel Belov, Peng Su, Peter Bak, Peter Bakkum, Peter Deng, Peter Dolan, Peter Hoeschele, Peter Welinder, Phil Tillet, Philip Pronin, Philippe Tillet, Prafulla Dhariwal, Qiming Yuan, Rachel Dias, Rachel Lim, Rahul Arora, Rajan Troll, Randall Lin, Rapha Gontijo Lopes, Raul Puri, Reah Miyara, Reimar Leike, Renaud Gaubert, Reza Zamani, Ricky Wang, Rob Donnelly, Rob Honsby, Rocky Smith, Rohan Sahai, Rohit Ramchandani, Romain Huet, Rory Carmichael, Rowan Zellers, Roy Chen, Ruby Chen, Ruslan Nigmatullin, Ryan Cheu, Saachi Jain, Sam Altman, Sam Schoenholz, Sam Toizer, Samuel Miserendino, Sandhini Agarwal, Sara Culver, Scott Ethersmith, Scott Gray, Sean Grove, Sean Metzger, Shamez Hermani, Shantanu Jain, Shengjia Zhao, Sherwin Wu, Shino Jomoto, Shirong Wu, Shuaiqi, Xia, Sonia Phene, Spencer Papay, Srinivas Narayanan, Steve Coffey, Steve Lee, Stewart Hall, Suchir Balaji, Tal Broda, Tal Stramer, Tao Xu, Tarun Gogineni, Taya Christianson, Ted Sanders, Tejal Patwardhan, Thomas Cunninghman, Thomas Degry, Thomas Dimson, Thomas Raoux, Thomas Shadwell, Tianhao Zheng, Todd Underwood, Todor Markov, Toki Sherbakov, Tom Rubin, Tom Stasi, Tomer Kaftan, Tristan Heywood, Troy Peterson, Tyce Walters, Tyna Eloundou, Valerie Qi, Veit Moeller, Vinnie Monaco, Vishal Kuo, Vlad Fomenko, Wayne Chang, Weiyi Zheng, Wenda Zhou, Wesam Manassra, Will Sheu, Wojciech Zaremba, Yash Patil, Yilei Qian, Yongjik Kim, Youlong Cheng, Yu Zhang, Yuchen He, Yuchen Zhang, Yujia Jin, Yunxing Dai, and Yury Malkov. **GPT-4o System Card**, 2024. 19

[37] METR. **Details about METR's preliminary evaluation of GPT-4o — metr.github.io**. `https://metr.github.io/autonomy-evals-guide/gpt-4o-report/`. [Accessed 02-03-2025]. 19

[38] MRBULLWINKLE. **Azure OpenAI Service documentation - Quickstarts, Tutorials, API Reference - Azure AI services — learn.microsoft.com**. https://learn.microsoft.com/en-us/azure/ai-services/openai/. 19

[39] DAYA GUO, QIHAO ZHU, DEJIAN YANG, ZHENDA XIE, KAI DONG, WENTAO ZHANG, GUANTING CHEN, XIAO BI, Y. WU, Y. K. LI, FULI LUO, YINGFEI XIONG, AND WENFENG LIANG. **DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence**, 2024. 19

[40] **GitHub - devmentors/Pacco: Sample microservices solution in .NET Core 3.1 based on the cloud-agnostic approach — github.com**. https://github.com/devmentors/Pacco. 20

[41] **GitHub - apache/brpc: brpc is an Industrial-grade RPC framework using C++ Language, which is often used in high performance system such as Search, Storage, Machine learning, Advertisement, Recommendation etc. "brpc" means "better RPC". — github.com**. https://github.com/apache/brpc. 20

[42] **GitHub - microsoft/cpprestsdk: The C++ REST SDK is a Microsoft project for cloud-based client-server communication in native code using a modern asynchronous C++ API design. This project aims to help C++ developers connect to and interact with services. — github.com**. https://github.com/microsoft/cpprestsdk. 20

[43] TIANYI ZHANG, VARSHA KISHORE, FELIX WU, KILIAN Q. WEINBERGER, AND YOAV ARTZI. **BERTScore: Evaluating Text Generation with BERT**. *CoRR*, **abs/1904.09675**, 2019. 25

[44] ALIREZA SALEMI, EMAD KEBRIAEI, GHAZAL NEISI MINAEI, AND AZADEH SHAKERY. **ARMAN: Pre-training with Semantically Selecting and Reordering of Sentences for Persian Abstractive Summarization**. In MARIE-FRANCINE MOENS, XUANJING HUANG, LUCIA SPECIA, AND SCOTT WEN-TAU YIH, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9391–9407, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. 25

[45] Alexander R. Fabbri, Wojciech Kryscinski, Bryan McCann, Caiming Xiong, Richard Socher, and Dragomir R. Radev. **SummEval: Re-evaluating Summarization Evaluation**. *CoRR*, **abs/2007.12626**, 2020. 25

[46] Xiaobo Guo and Soroush Vosoughi. **Length Does Matter: Summary Length can Bias Summarization Metrics**. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 15869–15879, Singapore, December 2023. Association for Computational Linguistics. 26

[47] Daniel Deutsch, Rotem Dror, and Dan Roth. **A Statistical Analysis of Summarization Evaluation Metrics using Resampling Methods**. *CoRR*, **abs/2104.00054**, 2021. 26

Appendix